Teaching embedded software development utilising QNX and Qt with an automotive-themed coursework application

Barrie, Peter; Morison, Gordon

Published in:
2014 6th European Embedded Design in Education and Research Conference (EDERC)

DOI:
10.1109/EDERC.2014.6924348

Publication date:
2014

Document Version
Author accepted manuscript

TEACHING EMBEDDED SOFTWARE DEVELOPMENT UTILISING QNX AND Qt WITH AN AUTOMOTIVE-THEMED COURSEWORK APPLICATION

Peter Barrie and Gordon Morison

School of Engineering and Built Environment, Glasgow Caledonian University
Cowcaddens Road, G4 0BA, Glasgow, Scotland, UK
phone: + (44)141-331-3025, fax: + (44)141-331-3370, email: peter.barrie@gcu.ac.uk
web: www.gcu.ac.uk

ABSTRACT

Within the later stages of many Electronics or Computer Science degree programmes it is common to cover the programming of real-time systems, sometimes with an embedded orientation, instructing students in the fundamentals and application of multitasking and multithreading. Within this paper we detail the approach taken at Glasgow Caledonian University in the design of such a module as part of a programme in Computer and Electronic Systems Engineering. In particular we explain how we structure laboratory exercises to reflect the typical industrial design practice of partitioning systems into application layer and presentation layer - utilising QNX ™ RTOS for application, integrated with open-source Qt for presentation and human interaction. The target application is based on a simple automotive dashboard executing on the Beagleboard-xM platform. The approach is shown to raise student interest and understanding of embedded application-building using highly productive toolsets.

1. INTRODUCTION

A typical undergraduate electronics-based programme will utilise a range of programming languages but typically C/C++ might be utilised as a key language family for the programming of embedded systems. As students apply and integrate their newly developed programming skills within project work they naturally discover that the single-threaded programming paradigm does not always deliver an ideal model for mapping onto a number of concurrent system activities, especially where the software has to be time-sensitive and be able to react to multiple real-world events. They also discover problems in data sharing (mutual-exclusion issues) and issues of prioritising code to match the differing time-constraints of the application. For a well-rounded programme in the area of computer and electronic systems we feel it is important to provide students with experience in advanced programming practices so that they can learn and apply appropriate software models for the creation of software to implement well-structured reactive embedded systems. The move towards integrating more advanced programming practices in embedded education is well recognised [1]. Our chosen route involves the delivery of a taught module that includes the application of the powerful QNX [2] real-time operating system (RTOS) and development environment with consequential educational benefits that are discussed below.

In designing such a module we also feel it is essential to attempt to maximise student engagement with coursework by providing an interesting and industry-relevant application environment, so the major coursework exercise within the module includes a graphical automotive dashboard and the creation of associated application software. In implementing such a system we also wish to reflect key real-world architectural choices such as the separation of user-interface from the underlying application software; in order to facilitate this we have included an open-source Qt layer for the students to interact with; Qt delivers the presentation layer to support the building of the automotive human interface.

The laboratory classes are based on the following development environment: 1. BeagleBoard-xM Rev C (TI DM3730, ARM Cortex A8 core). 2. QNX Neutrino RTOS V6.5.0 SP1. 3.Code development: QNX Momentics V. 4.7.

2. SELECTION OF TEACHING RTOS

2.1 Selection Criteria

The underpinning philosophy is to deliver an academic module that has the following student learning outcomes:

- To understand the fundamental concepts of multitasking software within the context of a real-time operating system: processes and threads, communication, synchronisation, mutual-exclusion, timers, implementing ISRs. To understand and apply the concepts of scheduling tasks on a finite CPU resource.
- To understand and implement software patterns to support the design of software in a multitasking context. Increase knowledge of C programming.
- To understand the importance of standardisation for portability (using POSIX standard). Be able to confidently implement applications that utilise POSIX Thread Programming (pthreads), POSIX timers, messaging, shared-memory, mutex, semaphores and condition-variables.
- To be able to utilise appropriate toolsets to implement, debug, monitor and characterise the performance of applications under an RTOS at a deep level.

- To be able to understand and implement a software structure that separates the human interface from the main application. To gain experience with a widely-used and representative API to deliver this presentation layer – in this case Qt.

In selecting an appropriate operating-system environment for the module the following criteria were judged as being important:

- The kernel should be designed principally as real-time rather than a general-purpose kernel and should have a POSIX API available.
- There must be a wide range of target platforms with available board-support-packages. In our case we specifically wanted current and future support for the Beagleboard range of boards.
- There must be a standard IDE framework, such as Eclipse. There should be a built-in debugger that supports multithreading.
- The OS kernel should be capable of being instrumented so that detailed multitasking behaviour can be logged. There must be a tool integral to the IDE that can provide a graphical-user-interface to represent all multitasking behaviour on a timeline. There should also be tools to show detailed memory behaviour and also application performance profiling. Other tools such as code-coverage are useful.
- The environment should be as fully integrated as possible, not built from multiple installs since this is time-consuming and the resulting environment could lack cohesion. The appropriate environment will supply everything within a single IDE. Additionally, the documentation for the entire system being in a single environment will also be very useful.
- Another (major) issue in selecting a new software environment within an educational environment is cost. We wish to have low or zero licence cost.

Many of these criteria can be met from open-source toolsets and some real-time flavours of Linux could be possible candidates. However, in creating and maintaining a teaching laboratory it is important to be able to deliver a highly reliable environment in a short space of time with the toolset being as fully integrated as possible. It is important that the usability of the environment be as high as possible to minimise the possibility of issues that detract from the student learning experience. Creating development environments from multiple open-source packages can often lead to (time-consuming) complications and this led the authors to look at more fully integrated commercial offerings. From previous experience we have found that environments that fully meet the criteria above are made available from some major RTOS vendor and their offerings include free educational licences. Two such vendors include Wind River and QNX Software Systems. The principal author has good experience of using RTOS and toolset offerings from both companies and has used the QNX Neutrino kernel within industrial projects and industrial taught courses as well as previous undergraduate teaching. There is a very active community portal (*Foundry27*) made available for QNX with helpful and active forums and many new operating system ports made freely available. Additionally, students can access a one-year QNX development license for personal use. All these factors have influenced the choice of QNX as the chosen development route.

2.2 Flexibility of Development

Another key aspect is the requirement for flexible development. There are time-pressures on most student laboratory accommodation that can limit student access to development environments and consequently we wish to support more flexible practices that allow students to work (off-site) in their own time with as few technical limitations as possible. Our choice of development routes supports this and this is explained in section 3.2

3. THE DEVELOPMENT ENVIRONMENT

3.1 Standard Environment

The development environment includes The QNX Momentics IDE running on a host Windows 7 PC connected to one or more target development boards via two connections: 1. Serial (for a basic TTY interface and access to the target bootstrap messages) and 2. Once the target is booted then an Ethernet connection is available delivering a high-speed connection between host and target. Figure 1 shows the standard development configuration

The Beagleboard target provides peripheral I/O for coursework applications. For our automotive application we make use of a DVI interface for an LCD display and utilise GPIO to interface to a simulated pulse-based vehicle speed sensor. USB supports an input from a touch-screen or mouse. To lower the costs we have utilised a standard LCD monitor and mouse to represent a touch screen. QNX provides various graphical options; we are using OpenGL® ES: a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems [3] that is fully configured as part of the Beagleboard Board Support Package (BSP). This in turn supports the Qt presentation layer.

3.2 Flexible Working Environment

To support students working outside of the laboratory we provide a loan scheme for the target development boards. To make best use of the development environment a student will need to use the Ethernet port on their home machine to connect to the target board and have access to a spare LCD display (the display is required only for user-interactive applications).

In order to deliver an alternative simplified target environment and provide a high degree of flexibility we

provide a secondary target where students can work without a physical board but still undertake a significant proportion of the development work. This is based on a VMWare QNX Neutrino target image available (x86 target); with QNX Momentics IDE running under Windows (XP and Windows 7 have been tested) and students can connect to the VMWare-hosted target image running QNX Neutrino kernel on the same PC. The result is a complete host and target setup running on one PC or laptop, delivering a very convenient environment for experimentation. This includes the graphical user interface (GUI) required for the automotive application. What is not provided by this setup is exact real-time since it runs under emulation; additionally, this setup lacks the physical I/O available on the Beagleboard. However, this option has proved to be an extremely useful learning environment, with many of the coursework exercises (that do not depend on exact temporal issues) being suitable to run under emulation, with no difference in the student learning experience. Figure 2 shows the virtualised configuration.

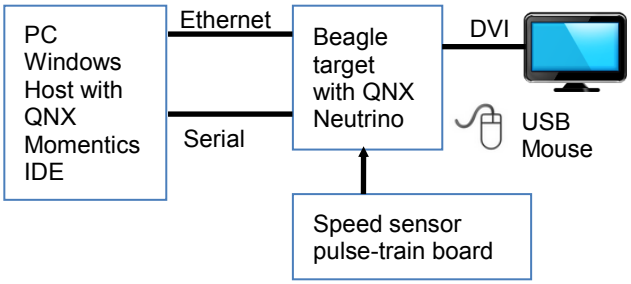

Figure 1 – Standard Development Configuration

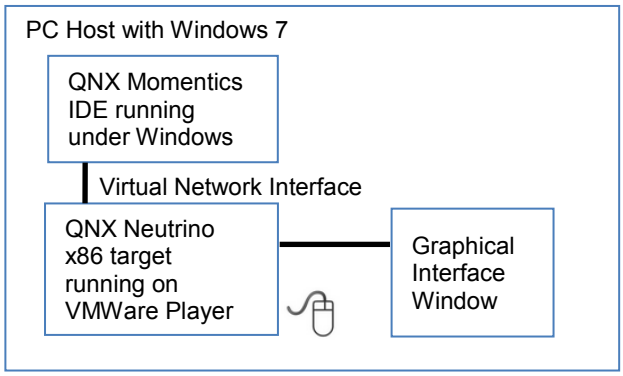

Figure 2 – Virtualised Development Configuration

4. LABORATORY SCHEDULE

The laboratory sessions are delivered over a twelve-week period. There are a number of fundamental concepts that must be assimilated prior to students being able to attempt the final automotive-flavoured coursework exercise. Within each laboratory session the students are led through a new concept (*programming* and *development tools* exercise are delivered) and provided with some demonstration code for initial experimentation. Students then have to attempt design, implementation and test of a set of coursework exercises that they have to complete and submit for the following week. In order of presentation, the students undertake exercise with the following themes:

1. Debugging, command-line parsing, spawning processes under Neutrino, mounting file system.
2. Shell commands and scripting. Introduction to pthreads.
3. Multiple threads, prioritisation, RTOS event tracing and displaying behaviour, semaphores.
4. POSIX timers and associated data structures.
5. Code modularisation. Use of Mutex.
6. POSIX messaging for thread communication and synchronisation, ring buffers implementation.
7. Interrupts, shared memory objects, named semaphores, use of alarm timer functions, signal handling.
8. In weeks 8 to 12 the students undertake the major coursework exercise based on the automotive application.

This automotive coursework exercise and key aspects of implementation are described below.

5. THE AUTOMOTIVE SYSTEM

5.1 System Specification and Interfaces

This exercise is designed to deliver two learning outcomes: (a) to ensure that students can integrate their understanding of all the individual learning experiences to design, implement and test a complete system that includes user-interaction, and (b) to ensure that students understand the concept of separation of the *application layer* and *presentation layer* of a system and can undertake development that follows this pattern.

With this exercise the student is required to deliver a simple automotive dashboard that has animated graphic displays for vehicle speed, engine RPM, fuel level and includes a range of common warning icons. The dashboard provides user controls for indicators, hazard warning and system start/stop. These controls would ideally be activated by a touch-interface but are presently (for economy) implemented by a mouse interface. A pulse generator module is interfaced via a Beagleboard GPIO interrupt signal to provide a realistic simulation of the pulse train from a hall-effect sensor typically used to monitor wheel rotational speed. The engine RPM sensor signal is at present simulated, but could in future be provided in an identical manner to the wheel speed signals.

The students are given a specification that describes the main software components of the system and their application programming interfaces (APIs). The exercise is specifically designed to ensure that students must utilise all key concepts that they have learnt about in the introductory exercises. The students are free to design the system in any manner that they think is appropriate, based on the constraints of the specification. An initial Qt graphical interface is provided for the students; they are responsible for driving all aspects of the display and can alter its layout if they wish to. Figure 3 shows a view of the GUI with the

warning icons illuminated. This includes a coloured icon on the fuel gauge that represents fuel-warning levels (green, amber, red). A working odometer is also included.

The specification enforces the architecture shown in Figure 4 that will form part of the completed system. The architecture includes three software interfaces. The first two interfaces are used to drive the GUI: (1) A message-passing interface that receives *discrete* events as POSIX messages from a client process. These events are used by the GUI component to drive the discrete display elements, such as the warning icons and indicator displays. (2) A POSIX shared-memory interface where a client process can set the current values of *continuous* variables such as vehicle speed, RPM and fuel level.

The third interface yields GUI user-events to be used by a client process. This is implemented by POSIX message passing. So a user clicking on an active zone on the GUI will lead to GUI generation of an appropriate message.

The importance of the message passing and shared-memory interfaces is that they deliver a well-defined, standardised and narrow interface between the main application layer and the GUI. This enforces good design practices and also provides useful debugging points for the code that students will write to interface with the GUI.

Figure 3 The Graphical User Interface with Warning Icons and Indicators Above and User Controls Below

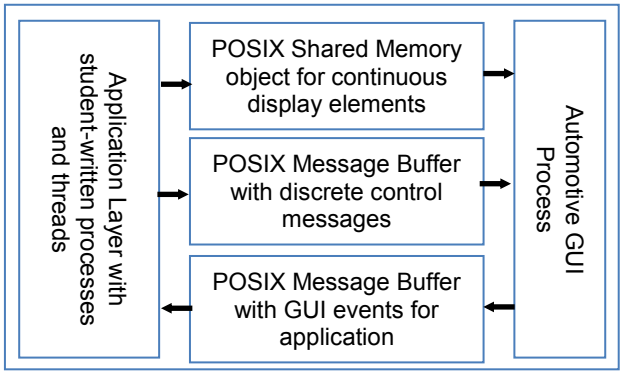

Figure 4 Interfaces between GUI and Application Layer

5.2 The GUI component

The GUI is built from the standard pattern that is used with Qt to interface with C/C++ code. Qt is a platform independent software toolkit for building UI's supports animation of visual objects and is implemented in C++; Qt interfaces are designed to be efficient and run well on embedded processors. As described in [4], with version 4.7, Qt includes a "declarative" language model. Writing a declarative UI is similar to describing a webpage with HTML – except the "markup language" for Qt Declarative is called "QML". QML describes the type and layout of GUI objects and their interaction with C++ applications. Interaction between the QML component and the C++ components is event-driven and uses a concept called "signals" and "slots". C++ application programs are designed to emit these "signals" (with a method call) to indicate the occurrence of specific events to a QML object (e.g. activate/hide/move a GUI object). When a QML application wishes to react to a user interface event (e.g clicking on an active zone) then appropriate QML code is written to directly call C++ interface methods called "slots" to execute appropriate actions when the user interface event is triggered. Signals and slots provide a well-defined interface between C++ and QML.

We have utilised this mechanism to build the automotive GUI component for the students to use in their application layer. The component is implemented as a QNX *process* (built and executed as one program) with three threads. Figure 5 shows this architecture; threads are notated in a dark colour.

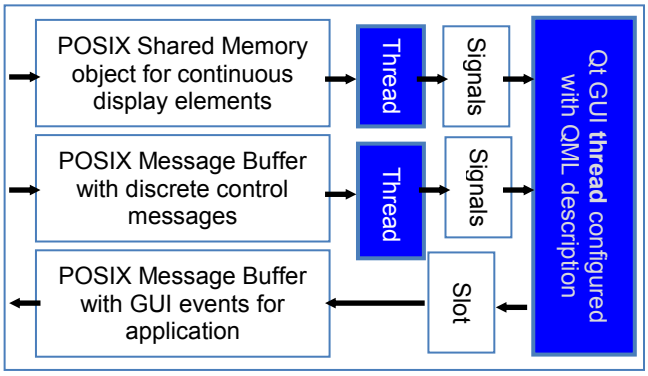

Figure 5 Automotive GUI Process Detail

- The first thread runs the Qt interface under the direction of the QML description (this thread is actually an automatic part of the a C++ programme built with Qt libraries). [Note: The QML code is saved as a file on the target file system and used by the Qt component of the application at run-time]. This thread is also responsible for invoking appropriate slot code when user-interface events are activated (clicking on user-interface objects) ; this code is used to write to the GUI events message-buffer. Messages posted here are to be serviced by the application layer that reads the posted POSIX messages.

- The second thread is used to read from the POSIX shared-memory (ten times per second) to identify updates from the application layer requiring display such as change of vehicle speed, RPM or fuel level); when there is a significant change in one of these values then the thread sends an appropriate signal to be handled by QML, updating the display.
- One final thread wakes up to service POSIX messages from the application layer with demands to update discrete elements such as warning indicators. On receiving a message the thread sends an appropriate signal to be handled by QML, updating the discrete elements of the display.

5.3 Application Layer

Students are responsible for designing, implementing and testing the application layer. The application has to include at least the following functionality: 1. Servicing a range of user-interface events and performing the corresponding control activities. 2. Servicing speed sensor interrupts to calculate vehicle speed. 3. Simulating fuel consumption and engine speed in RPM, 4. Sending updates to the GUI to control continuous and discrete GUI elements, 5. Maintaining system global-state to coordinate all activities.

The students have full control on whether they structure their code as one or more processes (a process being a container for one or more concurrent threads) and within each process then the number of threads required. They are encouraged to apply a modular approach to the creation of individual components and then undertake integration and final test. They are also encouraged to use the powerful tools available to analyse and refine the performance of the system.

6. RESULTS AND DISCUSSION

The module has been delivered to final year undergraduate students for the first time in the academic session 2013-14. The learning objectives are tested using formative and summative assessments. For the first eight weeks the students participate in the laboratories outlined in section 4 above, comprising set exercises. Students undertake this work within the laboratory session with support and individual formative feedback provided by the lab tutor. The tutor (through class-presentation and group discussion) remedies/clarifies any issues that manifest during this time and are common to several students. For each laboratory there is a summative assessment that is to be submitted for the start of the laboratory session in the following week. The lab tutor presents an outline solution for the previous exercise and then introduces the new laboratory work. The submitted exercises are assessed and marks and feedback sheets are returned to students. The sheets include feedback on technical and stylistic programming issues as well as a feedback on the quality of program documentation included.

Following the cycle of eight laboratories there is a four-week period devoted to the automotive application. Formal laboratory tutor support is provided during this time and formative feedback is provided to students as they evolve their designs. In practice continued support has been both necessary and valuable since students are assimilating a number of new concepts and also creating applications that are executing as multiple processes, each of which could have several threads. A degree of rigour is required in order to control the whole development process and tutor support is of high value at this time.

At the completion of this four-week development the students are individually assessed by submitted code and project report plus a demonstration of their system. The demonstration includes a number of specific tests to ensure that the completed system meets system functional and temporal requirements. There is also an oral component to this assessment so that students have to be able to demonstrate understanding of any aspect of their code design.

Success can be gauged in several ways. Firstly, all students managed to pass the module and the distribution of marks for the laboratory element of the assessment (50% of the total module mark is lab work) indicates that the students have gained a good understanding of the design and programming practices required for implanting systems with embedded multitasking software. Secondly, we had very good feedback from students indicating that they had a significant new learning experience that also integrated a number of software and hardware concepts from earlier taught modules. Students felt that the work extended their industry-relevant knowledge. Thirdly (and very significantly) our students are receiving good feedback (at job interviews) from potential (global) employers who are using similar techniques and can appreciate the value of what the students have learned.

5. ACKNOWLEDGMENTS

The authors thank the Computer and Electronic Systems Engineering students who participated in this class and are grateful for the support given by Texas Instruments; particularly the help provided by Djordje Marinkovic (Tl University Programme Manager).

REFERENCES

[1] “Managing the microprocessor course scope expansion.”. A Suyyagh, B Nahill, A Courtemanche, E Kirshin, Z Zilic, Boris Karajica. 2013 IEEE International Conference onMicroelectronic Systems Education (MSE).
[2] QNX RTOS: http://www.xnx.com
[3]OpenGL® ES https://www.khronos.org/opengles/
[4] “Qt HMI using QML” Tutorial: Creating a QNX UI with Qt Declarative. Dennis Kelly, QNX Software Systems. 18th May 2011. Available at: http://community.qnx.com